2018-12-17

Workflow Simulation Aware and Multi-Threading Effective Task Scheduling for Heterogeneous Computing

Kelefouras, Vasileios

10.1109/HiPC.2018.00032
2018 IEEE 25th International Conference on High Performance Computing (HiPC)
IEEE

Workflow Simulation Aware and Multi-Threading Effective Task Scheduling for Heterogeneous Computing

Vasilios Kelefouras Karim Djemame

***Abstract*—Efficient application scheduling is critical for achieving high performance in heterogeneous computing systems. This problem has proved to be NP-complete, heading research efforts in obtaining low complexity heuristics that produce good quality schedules. Although this problem has been extensively studied in the past, all the related works assume the computation costs of application tasks on processors are available a priori, ignoring the fact that the time needed to run/simulate all these tasks is orders of magnitude higher than finding a good quality schedule, especially in heterogeneous systems.**

In this paper, we propose two new methods applicable to several task scheduling algorithms for heterogeneous computing systems. We showcase both methods by using HEFT well known and popular algorithm, but they are applicable to other algorithms too, such as HCPT, HPS, PETS and CPOP. First, we propose a methodology to reduce the scheduling time of HEFT when the computation costs are unknown, without sacrificing the length of the output schedule (monotonic computation costs); this is achieved by reducing the number of computation costs required by HEFT and as a consequence the number of simulations applied. Second, we give heuristics to find which tasks are going to be executed as Single-Thread and which as Multi-Thread CPU implementations, as well as the number of the threads used.

The experimental results considering both random graphs and real world applications show that extending HEFT with the two proposed methods achieves better schedule lengths, while at the same time requires from 4.5 up to 24 less simulations.

***Keywords*-static task scheduling; simulation; multithreading; HEFT; Heterogeneity; multi-core;**

I. Introduction

A well-known strategy for efficient execution of an application on a heterogeneous computing environment is to partition the application into independent tasks and schedule such tasks over a set of available processors [1]. Normally, the application is represented as a Directed Acyclic Graph (DAG), which includes the characteristics of an application program such as the computation costs of the tasks, the data transfer time between tasks and task dependencies. The objective of the TS problem is to map the tasks on the (co)-processors and order their execution so that task precedence requirements are satisfied and a minimum schedule length is obtained (for the reminder of this paper we will refer to both processors and coprocessors as processors). This problem has proven to be NP-complete, even for the homogeneous case. Therefore, research efforts in this field have been mainly focused on obtaining low-complexity heuristics that produce good schedules [2], which is the topic of this paper.

In this paper, we propose two new methods applicable to several TS algorithms for Heterogeneous Computing Systems (HCS). We showcase both methods by using Heterogeneous Earliest Finish Time (HEFT) [3] algorithm, but this work is applicable to other algorithms too, such as Heterogeneous Critical Parent Trees (HCPT) [4], High-Performance Task Scheduling (HPS) [5], Performance Effective Task Scheduling (PETS) [1], Critical-Path-on-a-Processor (CPOP) [3] and others.

The first method (TSRS) reduces the scheduling time (the execution time for obtaining the output schedule) of HEFT when the computation costs are unknown. TSRS reduces the number of computation costs required by HEFT and therefore, the number of simulations required/performed, without sacrificing the length of the output schedule. Instead of simulating/running all tasks on every processor (to generate the DAG's computation costs) and then schedule the tasks (by using HEFT), we combine these two phases using an iterative approach. The second method (METS) provides low complexity heuristics finding which tasks are going to be executed as Single-Thread (ST) and which as Multi-Thread (MT) CPU implementations, as well as the number of threads used. The application tasks are assumed moldable [6] with the restriction that tasks can only be allocated to the physical cores of one CPU only.

The contributions are as follows: a) a novel TS methodology reducing HEFT's scheduling time, when the computation costs are unknown, b) low complexity heuristics considering tasks as both ST and MT CPU implementations, without requiring all the computation costs in the DAG, c) two methods applicable to several TS algorithms.

The evaluation of the proposed methods includes a large number of synthetic DAGs as well as five real world applications. The experimental results show that by using the proposed methods, HEFT provides better schedule lengths by facing tasks as both ST and MT implementations, while at the same time requires from 4.5 up to 24 less simulations.

In Section II, we introduce the TS problem. In Section III, the related work is reviewed. The proposed methods are given in Section IV, while the experimental results are discussed in Section V. Finally, Section VI is dedicated to conclusions and future work.

II. Task Scheduling Formulation

Resource model: The hardware (HW) platform consists of a set of p heterogeneous devices, either multi/single-core CPUs with m cores per CPU at maximum, or coprocessors (GPUs, FPGAs), that have diverse capabilities.

Workflow model: A workflow application is modeled as a DAG, G=(V,E), where V is the set of u nodes and each node $u_i \in V$ represents an application task. E is the set of e communication edges between tasks; each $e(i,j) \in E$ represents the task-dependence constraint such that task t_i should complete its execution before task t_j can be started [2] and is associated with a no negative weight value that represents the amount of data to be transmitted. The $n \times p$ computation cost matrix W stores the computation costs of the tasks, where n, is the number of the tasks; each element $w_{t,j} \in W$ refers to the estimated time to execute task t on processor p_j (please note that in the next paragraphs matrix W becomes $n \times p \times m$). The execution of any task is considered nonpreemptive. These model simplifications are common in this scheduling problem [2] [3] [7].

TSRS problem definition: This problem is the static scheduling of a single application, whose computation cost matrix W is unknown, in a set of p heterogeneous devices, in such a way that the scheduling length and the scheduling time (to deliver the output schedule), are minimized. Note that the scheduling time highly depends on the time needed to simulate the tasks.

Standalone TSRS assumes rigid (non-moldable) tasks (ST CPU implementations) and monotonic computation costs, e.g., consider a cluster with 2 type1 coprocessors, 2 type2 CPUs and 2 type3 CPUs, where ($w_{t1,type2} \geq w_{t1,type3} \geq w_{t1,type1}$) for task $t1$; then, we assume that ($w_{t,type2} \geq w_{t,type3} \geq w_{t,type1}$) for every task t. All the processors are classified into groups according to their computation capability (CC); a random task is run on every processor and the processor achieving the minimum execution time value is considered as the one with the highest CC; regarding multi-core CPUs, the CC refers to the one core only (ST implementations). All the groups are sorted in an increasing CC order, e.g., $proc\_order = (p_{type2}, p_{type3}, p_{type1})$. In case that the CC of two different processors is approximately the same, we can consider both in the same processor group. This procedure is not necessary for all the processors; however, in case we don't classify a processor into a group, all the computation costs on that processor must be known, increasing the number of simulations performed.

METS with TSRS problem definition: This problem is the static scheduling of an application consisting of a set of n moldable tasks, whose computation cost matrix W is unknown, in a set of p heterogeneous devices, in such a way that the scheduling time and scheduling length, are minimized. The application tasks are assumed moldable [6] with the restriction that tasks can only be allocated to the physical cores of one CPU only; moldable tasks are the tasks being allocated to a fixed number of processors before execution and stay unchanged afterwards, e.g., Pthreads, OpenMP. Thus, given a multi-core CPU with m cores, we consider every task as an m-thread implementation, where $m = [1, cores]$ and $cores$ is the number of CPU physical cores. The computation cost matrix W becomes $n \times p \times m$; if the processor is a coprocessor or a single-core CPU, m=1 ($w_{t,j,1}$). The core utilization factor is defined as, $factor_{t,j,m} = w_{t,j,1}/w_{t,j,m}$. We assume that first, the speedup function of the moldable tasks is non-decreasing [6] [8], i.e., $w_{t,i,f1} \leq w_{t,i,f2}$, where $f1 \succ f2$, $f1 \leq cores$, and second, every task scales equally in different CPUs ($factor_{t,i,f} = factor_{t,j,f}$). Thus, if $w_{t,i,1} \prec w_{t,j,1}$, then $w_{t,i,f} \prec w_{t,j,f}$ but not $w_{t,i,f1} \prec w_{t,j,f2}$, where $f1 \prec f2$.

Next, we present some common attributes used in TS problem, which we will refer to in the following sections.

Definition 1: $EST(t_i, p_j, m)$ denotes the Earliest Start Time (EST) of task t_i on processor p_j using m cores (for coprocessors or single-core CPUs, m=1) and defined as

$$EST(t_i, p_j, m) = \max\left\{T_{Avail}(p_j, m), T_{pred}(t_i, p_j)\right\}$$
$$T_{pred}(t_i, p_j) = \max_{t_l \in pred(t_i)}\{AFT(t_l) + c_{l,i}\} \quad (1)$$

where $T_{Avail}(p_j, m)$ is the earliest time at which the specific m cores of processor p_j are ready and $T_{pred}(t_i, p_j)$ is the time at which all data needed by task t_i arrive at the processor p_j. The communication cost $c_{l,i}$ is zero if the predecessor task t_l is assigned to processor p_j. For the entry task, $EST(t_{entry}, p_j) = 0$.

Definition 2: $EFT(t_i, p_j, m)$ denotes the Earliest Finish Time (EFT) of a task t_i on processor p_j using m cores:

$$EFT(t_i, p_j, m) = EST(t_i, p_j, m) + w_{t,j,m} \quad (2)$$

which is the EST of a task t_i on the specific m cores of processor p_j, plus the computation cost of an m-thread implementation of t_i on processor p_j. For the rest of this paper we will refer to $EFT(t_i, p_j, 1)$ as $EFT(t_i, p_j)$.

III. Related Work

To the best of our knowledge, all the TS algorithms assume the computation costs available a priori and there is no related work to TSRS. Moreover, there are no low complexity heuristics considering tasks as both ST and MT implementations. The second is close to the problem of scheduling moldable tasks with the restriction that tasks can only use the cores of one processor and most of them are based on a two-phase approach. First, the number of processors assigned for each task is selected and second, the rigid (non-moldable) tasks are scheduled by using a TS algorithm. In [6], they present a new algorithm combining dual approximation and ILP for moldable tasks on hybrid platforms of identical GPUs and CPUs. In [9] they present efficient algorithms for scheduling an application on hybrid platforms of identical CPUs and GPUs. In [10], they improve and compare two previous methods of theirs for Mixed-Parallel Applications on Heterogeneous Platforms. In [11], a

generic algorithm is presented with a performance guarantee for scheduling tasks with precedence constraints on CPU-GPU platforms. In [8], a new algorithm is proposed that supports arbitrary run-time functions of moldable tasks on identical processors. Compared to the aforementioned methods, METS achieves lower scheduling time, as first, the number of simulations required is lower and second METS complexity is low (equal to HEFT's).

Algorithm 1 HEFT Algorithm

1: Set the computation costs of tasks and communication costs of edges with mean values
2: Compute $rank_u$ for all tasks by traversing graph upward, starting from the exit task
3: Sort tasks in a scheduling list by decreasing order of $rank_u$ values
4: **while** there are unscheduled tasks in the list **do**
5: Select the first task, t_i, from the list for scheduling
6: **for** each processor p_j in the processor-set **do**
7: Compute $EFT(t_i, p_j)$ value using the insertion-based scheduling policy
8: **end for**
9: Assign task t_i to the processor p_j that minimizes EFT of task t_i
10: **end while**

Algorithm 2 HEFT with TSRS / HEFT with METS and TSRS

1: Sort in an increasing order all the groups of processors according to their computation capability (CC). Set the computation costs of tasks according to p_{ref} only ($w_{t,p_{ref},1}$) and the communication costs of edges with mean values
2: Compute $rank_u$ for all tasks by traversing graph upward, starting from the exit task
3: Sort tasks in a scheduling list by decreasing order of $rank_u$ values
4: **while** there are unscheduled tasks in the list **do**
5: Select the first task, t, from the list for scheduling
6: [$w_{t,i,1}()$, SL()]=**TSRS(t)**; / [$w_{t,i,m}()$, SL()]=**METS(t)**;
7: **for** each processor p_j in SL (simulation list) **do**
8: Compute $EFT(t_i, p_j)$ / $EFT(t_i, p_j, m)$ value with/without the insertion-based scheduling policy
9: **end for**
10: Assign task t_i to the processor p_j that minimizes EFT of task t_i
11: **end while**

Task scheduling (TS) can be performed at compile-time or at run-time, referred as static or dynamic scheduling. The static task scheduling algorithms are classified in two main categories. The first one includes algorithms that are based on heuristics, such as list scheduling [2] [3], clustering [12] or node duplication, while the second includes stochastic search algorithms, where the problem is modeled as an optimization problem using either ILP or CP models. Clustering heuristics are mainly proposed for homogeneous systems [12]. The duplication heuristics produce shorter makespans than list scheduling heuristics, but result in higher time complexity as well as to more processor availability and power [2]. List scheduling heuristics, on the other hand, produce the most efficient schedules, without compromising the makespan and with a low complexity [2]. Some of the most important list scheduling heuristics are: Predict Earliest Finish Time (PEFT) [2], HEFT [3], HCPT [4], HPS [5],PETS [1], Lookahead [13], Longest Dynamic Critical Path (LDCP) [7].

HEFT algorithm assumes rigid tasks and is shown in Algorithm 1; it has a prioritizing and a processor selection phase. In the first phase, task priorities are defined by using $rank_u$ which represents the length of the longest path from t_i to the exit node, including the computation cost of t_i and is given by $rank_u(t_i) = \overline{w_i} + \max_{t_j \in succ(t_i)}\{\overline{c_{(i,j)}} + rank_u(t_j)\}$. For the exit task, $rank_u(t_{exit}) = \overline{w_{exit}}$. The task list is ordered by decreasing value of $rank_u$. The task with the highest rank is scheduled first. In the processor selection phase, the task with the higher $rank_u$ value is assigned to the processor giving the EFT.

IV. Proposed TS Methodology and Heuristics

The two novel TS methods are given in Subsection IV-A and Subsection IV-B, respectively.

A. Task Scheduling methodology Reducing the number of task Simulations (TSRS)

This methodology consists of two stages, an initialization stage (line 1 in Algorithm 2), where all the processors are sorted in an increasing CC order (Section II) and the main stage. In Algorithm 2, we show HEFT with TSRS and HEFT with both METS and TSRS. The main stage of TSRS extends the processor selection phase, lines 6-8.

Initialization step: In line 1 (Algorithm 2), the DAG is initialized with the computation costs of the tasks on the one core of p_{ref} only (reference processor), i.e., $w_{t,p_{ref},1}$. Then, the generation of the other computation costs and the scheduling of the tasks are applied together, in an iterative approach; in line 6, TSRS discards the processors which cannot minimize the specific heuristic cost function used for the current task (regardless of their computation costs), while all the others are simulated and their computation costs are returned.

The DAG is initialized with the computation costs on the reference processor (p_{ref}) only and therefore the $rank_u$ values are no longer computed using the average costs but using the computation costs on p_{ref}, slightly affecting the task priority list; the priority list is not substantially affected because the computation costs are monotonic. In [14], the rank function of HEFT algorithm is investigated by using the mean, median, worst and best computation costs; it is shown that for random computation costs (not monotonic as in our case) first, different ways of computing $rank_u$ affects HEFT performance and second, the mean computation costs is not the best choice. In Subsection V-B1, we show that HEFT's schedule length is not degraded by TSRS and in addition to [14], we showcase that the mean computation costs do not provide better solutions than the p_{ref} ones. In terms of makespan, it is more efficient to select a Highest

Computational Capability Processor (HCCP) as p_{ref} (a last group processor). However, in METS (Subsection IV-B), p_{ref} cannot be a HCCP in all cases, because it has to be the multi-core processor containing the maximum number of cores (cannot be a coprocessor). Thus, given that TSRS is applied as both standalone method and together with METS, we will not consider p_{ref} as a fixed value.

Main Step: In this paper, we provide the TSRS without the insertion based scheduling policy as it is more comlex and the page size is limited. However, in Section 4 we have evaluated TSRS with and without the insertion policy.

The main step of TSRS (line 6 in Algorithm 2) reduces the number of candidate processors in the processor selection phase. The procedure follows. The EFT is given by Eq. 2 and consists of two parts, EST and $w_{i,j,f}$ (for coprocessors f=1). The second part of Eq. 2 ($w_{i,j,f}$) is an unknown value, as task t is not simulated on every processor group but on p_{ref} only, while the first part of Eq. 2 is known, as it refers to the processor availability time as well as to the finish time of the previously scheduled tasks. Given that first, the processor groups are sorted in an increasing CC order and second, the first part of Eq. 2 is known, we are able to reduce the number of candidate processors for task t, without excluding any processor with minimum EFT value. As an example, assume that the EFT values of t on 4 different single core processors are those in Eq. 3 and also $p_{ref} = p_3$.

$$\begin{aligned} EFT(t,p_1) &= w_{t,1,1} + 10 \\ EFT(t,p_2) &= w_{t,2,1} + 9 \\ EFT(t,p_3) &= 2 + 9 \\ EFT(t,p_4) &= w_{t,4,1} + 13 \end{aligned} \tag{3}$$

Given that ($w_{t,1,1} \geq w_{t,2,1} \geq w_{t,3,1} = 2 \geq w_{t,4,1}$), there is no need to simulate t on p_1 and p_2 as these two processors always give a larger EFT value than p_3 and therefore they will never be allocated for t by HEFT algorithm.

The proposed method is given in Algorithm 3. First, we compute the EFT values for all the processors by using $w_{t,p_{ref},1}$ instead of $w_{t,j,1}$ and put the minimum EFT value of every processor group i in $S(i)$ (lines 3-6). All the processors inside a group have identical computation costs.

In lines 8-16, we compare $S(i)$ with $S(j)$, where always holds ($i \succ j$) (and therefore always $w_{t,i,1} \leq w_{t,j,1}$). If the $EFT(t,i)$ value referring to processor group i is smaller or equal to any other $EFT(t,j)$ value to a slower group j, then j is not a candidate group and it is removed from the simulation list (SL). Let us follow the above example of Eq. 3 for lines 8-16 (Algorithm 3), where $w_{t,p_{ref},1} = 2$ and thus $EFT(t,p_1) = 12$, $EFT(t,p_2) = 11$, $EFT(t,p_3) = 11$, $EFT(t,p_4) = 15$). First, the $EFT(t,p_4)$ value is compared to $EFT(t,p_3)$,$EFT(t,p_2)$ and $EFT(t,p_1)$ but the *if-condition* in line 12 is never true. Then, the $EFT(t,p_3)$ value is compared to $EFT(t,p_2)$ and $EFT(t,p_1)$ and because $EFT(t,p_2)$ and $EFT(t,p_1)$ give larger or equal values, they are both excluded from SL etc. Thus, the processor groups with $j = 1$ and $j = 2$ are removed from the list. The number of candidate processors is reduced without excluding any processors with minimum EFT value.

Algorithm 3 TSRS without using the insertion based scheduling policy

```
1: [w_{t,i,1}(), SL()] = TSRS (t) {
2:
3: for (i = 1, Proc.groups) do
4:    compute EFT(t, j) for every p_j in group i, by using
      w_{t,i,1} = w_{t,p_ref,1}
5:    Put the min EFT(t, j) value from every processor group i
      in S(i)
6: end for
7:
8:          /*Reduce the search space*/
9: Put all processor groups in the simulation list (SL)
10: for (i = Proc.groups, 2, −1) do
11:    for (j = i − 1, 1, −1) do
12:       if (S(i) ≤ S(j)) then
13:          remove processor group j from SL
14:       end if
15:    end for
16: end for
17:             /*this step is optional*/
18: if (p_ref ∉ HCCP group) then
19:    for (i = 1, Proc.groups − 1) do
20:       if (S(i) ≤ min_EFT_on_p_HCCP) then
21:          remove p_HCCP group from SL
22:       end if
23:    end for
24: end if
25:
26: Get the w_{t,i,1} values that i ∈ SL (if any) //t is simulated
27: Return w_{t,i,1}(), SL() }
```

In case that ($p_{ref} \in HCCP$ group), the lines 18-24 in Algorithm 3 are omitted. On the other hand, when p_{ref} is not a HCCP, the method given in lines 8-16 (Algorithm 3) is not able to reduce the number of simulations on the HCCP group. To do so, we have to define a lower bound value regarding how fast the HCCP is. We define a very low unreachable lower bound value on the HCCP, e.g., task t will never run 50 times faster than p_{ref} ($w_{t,p_{ref},1}/50 \leq w_{t,p_{HCCP},1} \leq w_{t,p_{ref},1}$) for every task t. This procedure is given in lines 18-24 in Algorithm 3; if $S(i)$ (where $i \prec Proc.groups$ - $Proc.groups$ is the last group, HCCP group) is lower or equal to the minimum $EFT(t,HCCP)$ value that the HCCP group can get, then the HCCP group is removed from SL. Let us follow the previous example (Eq. 3), where the method given in lines 8-16 (Algorithm 3) has already excluded p_1 and p_2 from SL. If we apply the method given in lines 18-24 (Algorithm 3) with ($min\_EFT\_on\_p_{HCCP} = 2/50 + 13$), then the minimum value that p_4 can get is always larger than $EFT(t,p_3)$ and thus p_4 is also excluded from SL.

However, the procedure in lines 18-24 slightly degrades HEFT's output schedule length because we do not know how larger the $w_{t,i,1}$ values can be in comparison with $w_{t,p_{ref},1}$.

Let us give an example, consider we have to compute the EFT values of t on 4 different single core processors and p_3 is the p_{ref}. Moreover, consider that Eq. 2 gives the following:

$$\begin{aligned} EFT(t,p_1) &= w_{t,1,1} + 9 \\ EFT(t,p_2) &= w_{t,2,1} + 9 \\ EFT(t,p_3) &= 2 + 15 \\ EFT(t,p_4) &= w_{t,4,1} + 13 \end{aligned} \tag{4}$$

The lines 8-16 in Algorithm 3 exclude p_1 and p_3 from SL. The lines 18-24 (Algorithm 3), with $(min\_EFT\_on\_p_{HCCP} = 2/50 + 13)$, exclude p_4 from SL, meaning that t is assigned on p_2, which is not always the processor with the minimum EFT (it depends on the $w_{t,2,1}$ value). We know that $(w_{t,2,1} \geq 2)$, but we don't know how large $w_{t,2,1}$ is; thus, if $(w_{t,2,1} + 9 \succ (2/50 + 13))$ and therefore $(w_{t,2,1} \succ 2/50 + 4)$, then t may run faster on p_4 than on p_2, and in that case, it shouldn't have been removed from the list. In that case, the more the processor groups, the more the makespan degradation. However, the above refer to special cases only and therefore the makespan degradation is very low. This step (lines 17-24) is optional.

At last, t is simulated on all the processors in SL (line 26) and the computation costs are returned (line 27).

TSRS does not increase HEFT's complexity which remains $O(e \times p)$; nevertheless, the algorithm's complexity is undermined as normally, the time needed for the task simulations to be performed is much higher (line 6).

TSRS is applicable to most of the TS heuristics using the minimum EFT value as the heuristic cost function, such as HCPT [4], HPS [5], PETS [1], CPOP [3] [15] list scheduling algorithms, [12] [16] clustering algorithms, and others.

B. Multi-Threading Effective Task Scheduling heuristics (METS)

In this paper, METS is applied together with TSRS, in order to optimize for both scheduling time (TSRS) and length (METS). Unlike standalone TSRS, where every processor group has a unique computation cost, in this case a CPU group has as many computation costs as its number of cores ($cores$); thus, standalone METS requires 'cores' computation costs for every CPU and one for every coprocessor. However, when METS is applied together with TSRS, we apply only one simulation for every processor group, leveraging the fact that every task scales equally in different CPUs ($factor_{t,i,f} = factor_{t,j,f}$) and that the scaling function is non decreasing (Section 2). Thus, standalone METS, where the computation cost matrix is known, provides better scheduling lengths.

The new version of TSRS is given in Algorithm 4 and is similar to Algorithm 3. The $EFT(t,j,1)$ values for the ST implementations are computed as in Algorithm 3, while the $EFT(t,j,f)$ values for the MT implementations are computed by using median core utilization factor values, $(fact. = 1.5, 2, 2.8, 3, 3.5)$ for $(2, 3, 4, 5, 6)$ cores, respectively (line 5 in Algorithm 4). Using median core utilization factor values is a good heuristic for line 13, as we assume that the tasks scale equally among different CPUs. The best ST and MT EFT value for each processor group is stored into $S(i)$ and $M(i)$, respectively. It is important to note that a) the best MT EFT value is not always the one using the maximum number of threads and b) the MT EFT value is not always smaller than the ST EFT, e.g., consider the case where the five out of six cores are not available in the near future. Lines 10-17 in Algorithm 4, are similar to lines 8-16 in Algorithm 3, but in Algorithm 4 a processor group is removed from SL if both the best ST and MT values are larger than those of another group.

Algorithm 4 TSRS (Algorithm 3) when it is called by METS (without insertion based scheduling policy)

```
1: [SL(), S(), M()] = TSRS (t) {
2:
3: for (i = 1, Proc.groups) do
4:     compute EFT(t, j, 1) for every p_j in group i, by using
       w_{t,i,1} = w_{t,p_ref,1}
5:     compute EFT(t, j, f) for every p_j in group i and for all
       the thread combinations f, by using w_{t,i,1} = w_{t,p_ref,1} and
       w_{t,i,f} = w_{t,p_ref,1} × fact.(f)
6:     Put the min EFT(t, j, 1) and EFT(t, j, f) values from
       every processor group i in S(i) and M(i), respectively
7: end for
8:
9:          /*Reduce the search space*/
10: Put all the processor groups in the simulation list (SL)
11: for (i = Proc.groups, 2, −1) do
12:     for (j = i − 1, 1, −1) do
13:         if (min(S(i), M(i)) ≤ min(S(j), M(j))) then
14:             remove processor group j from SL
15:         end if
16:     end for
17: end for
18:
19: if (p_ref ∉ HCCP group) then
20:     for (i = 1, Proc.groups − 1) do
21:         if (S(i) ≤ min_EFT_on_p_HCCP) then
22:             remove p_HCCP group from SL
23:         end if
24:     end for
25: end if
26: Return SL(), S(), M() }
```

The key points of METS are the following:

1) ST implementations are more efficient for tasks with high Communication to Computation Ratio (CCR) values
2) MT implementations are more efficient when the task parallelism is low
3) When the task parallelism is high, ST/MT implementations are more efficient when the range of $w_{t,i,f}$ values among different tasks, is low/high, respectively.
4) We use $factor_{t,i,f1}$ value to update $factor_{t,j,f2}$, where $f1 \succ f2$

Regarding the first key point, ST implementations are more efficient for high CCR values. The data transfer cost

is minimized when the tasks are executed on the same processor as the data remain in the processor's disk/memory. The more tasks each processor can handle in parallel, the less the communication cost, as the intra-processor transfer cost is very low. The *if-condition* in line 11 (Algorithm 5) implements the above idea. By using a ST implementation for a parent task that gives too much data to its children, we reduce the probability of its children tasks to get data from another processor(s). On the other hand, by using a ST implementation for a child task which gets too much data from its parents, we increase the probability of the other children (with the same parents) to be assigned to the same processor and therefore minimize the transfer cost.

As far as the second key point is concerned, when the number of the ready tasks is smaller than the number of the processors, there is no reason to save any cores, and thus the implementation giving the minimum EFT value is selected, no matter the number of cores used (the implementation giving the minimum EFT is not always MT). The *if-condition* in line 14 (Algorithm 5) implements the above idea. In Algorithm 5, ST&MT means that we seek for the solution giving the minimum EFT value no matter the number of threads/cores used (either ST or MT). This heuristic does not hold for high CCR values for the reason explained in the previous paragraph and therefore, the 'else if' condition in line 14.

Let us explain the second key point further, consider there are four identical multi-core processors and only 4 ready tasks. In that case, it is not efficient to save any cores and therefore MT implementations for all the tasks is the best solution no matter the number of threads used. However, if there are 5 ready tasks, it might not be efficient to use MT implementations for all the tasks, because other thread combinations have to be investigated too. This is why we have used the 'Threshold' value in line 8 (Algorithm 5), indicating the number of ready tasks should exist in order to use ST&MT implementations; in this case, the 'Threshold' value in line 8 is ($Threshold = 4$). Keep in mind that MT refers to the best MT solution, no matter how many threads are used. Now consider the case that there are 5 ready tasks and a HW environment with three identical multi-core processors and one GPU (let us assume that the tasks run two times faster on the GPU). One could think that it is not efficient to use MT implementations for all the multi-core processors because one ready task will have to wait until another finishes its execution. However, if the tasks are executed 2 times faster on the GPU than on the processor, the GPU will have executed 2 tasks until the three processors finish their execution. Thus, the GPU 'counts' for 2 processors and there is no reason to save any cores. In this case, ($Threshold = 5$) and not ($Threshold = 4$). The '$Threshold$' value depends on a) the number of the processors, b) the number of the cores each processor has, c) how faster/slower is one processor to another. The '$Threshold$' value is application independent and depends solely on the HW infrastructure. Thus, it can be found 'off-line'. In Section 4, ($Procs \leq Threshold \prec 2 \times Procs$), where $Procs$ is the number of the processors.

Regarding the third key point above, i.e., when the number of ready tasks is larger than the '$Threshold$' value, the MT implementations are efficient only in the case that the range of the $w_{t,i,j}$ values for different tasks t is high and in particular for the tasks having larger $w_{t,i,j}$ values than the others. This is because the core utilization factor value is always lower than the number of cores and therefore the time needed for a task to be executed as an f-thread implementation is always larger than executing f different tasks. Let us give an example, consider 8 identical tasks ready for execution and two identical 4-core processors. Also consider that the eight tasks need $(10, 6, 4, 3)$ secs to be executed, using $(1, 2, 3, 4)$ threads, respectively. If all the tasks are considered as ST, then 10 secs are required for them to be executed. On the other hand, by using 4-thread or 2-thread implementations only, 12 secs are needed. However, if half of the tasks need $(15, 9, 6, 4.5)$ and the other half $(10, 6, 4, 3)$ seconds to be executed by using $(1, 2, 3, 4)$ threads, respectively, then using only ST implementations is not the best option. If we run the heavy tasks as 4-thread implementations and the light ones as ST ones, then the overall execution time is 14.5 secs, while by using ST only, it is 15 secs. The *if-condition* in line 21 (Algorithm 5) satisfies that only the tasks with high $w_{t,i,j}$ values are considered as MT. If a task's rank value is larger than 1.3 times the minimum rank value of C (the tasks that are going to be executed in the near future), it is further processed as an ST&MT implementation, otherwise it is assigned as a ST.

In contrast to line 16, where an ST&MT implementation is always selected regardless of whether t is effectively split into multiple threads or not, in line 22, the number of tasks waiting for execution is higher than the number of processors and thus we have to consider the scenario that t may give a low core-utilization factor. Thus, we get $w_{t,p_{ref},f}$ value, where f is the maximum number of threads in SL, and compute the utilization factor. If the factor is large enough, we use a ST&MT implementation, otherwise, we give a second chance for t to be executed with fewer threads, i.e., $\lceil f/2 \rceil$ (line 29). The good utilization factor values used are (1.6,2.35,3.4,3.9,4.7) for (2,3,4,5,6) threads, respectively.

Regarding the fourth key point above, we assume that $factor_{t,i,f} = factor_{t,j,f}$, where i, j are multi-core processors. Moreover, we measure $factor_{t,i,f1}$ and update $factor_{t,j,f2}$, where $f1 \succ f2$; $factor_{t,j,f2} = (f2 \times factor_{t,i,f1})/f1$. This procedure is applied in lines 26 and 33 (Algorithm 5) in order to update the EFT values on the other processors according to $factor_{t,p_{ref},f}$ value.

METS is given in Algorithm 5. All the coefficients found experimentally. First, TSRS finds the candidate processors for task t (line 3). If there is no multi-core candidate

Algorithm 5 METS with TSRS

[$\mathbf{w_{t,i,thr}}$(), SL()] = METS (t) {

[SL(),S(),M()]=TSRS(t);
if (SL contains no multi-core processor) **then**
  Get the $w_{t,i,1}$ values that $i \in SL$ (if any) - $thr = 1$
else
  A $\leftarrow$ next 6 ready tasks
  B $\leftarrow$ next 'Threshold' tasks
  C $\leftarrow$ ready tasks that $(Rank_u \succ 0.7 \times Rank_u(t))$ /*tasks to be executed in the near future only*/

  if (at least half of the tasks in A contain an edge (either parent or child edge), where $c_{n,m}/w_{t,p_{ref},1} \geq 1.5$) **then**
    /*processors are faced as ST only*/
    [$\mathbf{w_{t,i,1}}$(), SL()] = kernel (ST,t);
  else if (at least one task in B is not ready) **then**
    /* Task parallelism is low. Use the implementation giving the min EFT, no matter the # of the threads*/
    [$\mathbf{w_{t,i,thr}}$(), SL()]= kernel (ST&MT,t);
  else
    /* task parallelism is high */

    /*if the range of $w_{t,p_{ref},1}$ values among diff. tasks is high*/
    if $(Rank_u(t) \succ (1.3 \times min(Rank_u(C)))$ **then**
      Get $\mathbf{w_{t,p_{ref},f}}$, where f is the max number of threads in SL
      $factor_{t,p_{ref},f} = w_{t,p_{ref},1}/w_{t,p_{ref},f}$
      if $(factor_{t,p_{ref},f} \succ good.factor(f))$ **then**
        /*Use the implementation giving the min EFT, no matter the # of the threads*/
        Use $factor_{t,p_{ref},f}$ to update EFT to other procs
        [$\mathbf{w_{t,i,thr}}$(), SL()] = kernel (ST&MT,t);
      else
        Get $\mathbf{w_{t,p_{ref},\lceil f/2 \rceil}}$
        $factor_{t,p_{ref},\lceil f/2 \rceil} = w_{t,p_{ref},1}/w_{t,p_{ref},\lceil f/2 \rceil}$
        if ($(factor_{t,p_{ref},\lceil f/2 \rceil} \succ good.factor(\lceil f/2 \rceil))$ AND $((\lceil f/2 \rceil) \succ 1)$) **then**
          /*Use the implementation giving the min EFT, no matter the # of the threads*/
          Use $factor_{t,p_{ref},\lceil f/2 \rceil}$ value to update EFT to other processors
          [$\mathbf{w_{t,i,thr}}$(), SL()] = kernel (ST&MT,t);
        else
          /*processors are faced as ST only*/
          [$\mathbf{w_{t,i,1}}$(), SL()] = kernel (ST,t);
        end if
      end if
    else
      /*processors are faced as ST only*/
      [$\mathbf{w_{t,i,1}}$(), SL()] = kernel (ST,t);
    end if
  end if
end if
Return $w_{t,i,thr}$(), SL() }

[$\mathbf{w_{t,i,thr}}$(), SL()] = kernel (T,t) {
if $(T == ST)$ **then**
  [SL(),S(),M()] = TSRS(t) - by using S() only, not M()
  Get the $w_{t,i,1}$ values that $i \in SL$ (if any) - $thr = 1$
else
  [SL(),S(),M()] = TSRS(t)
  Get the $w_{t,i,thr}$ values (if any) where $i \in SL$ and thr is the number of threads of the $\min(S(i), M(i))$
end if
Return $\mathbf{w_{t,i,thr}}$(), SL() }

processor, the procedure is trivial. Otherwise, the multiple *if-conditions* take place finding whether the selected processor will use a ST or a ST&MT implementation. In the case that a ST is selected, we simulate t as ST only. Otherwise, if a ST&MT is selected, we simulate t on the remaining processors but t is simulated either as ST or MT, not both.

The heuristics presented in Subsection IV-B, can be applied together with TSRS to the algorithms that TSRS is applicable to. Moreover, METS can be applied as a standalone method too.

In terms of time complexity, METS gives $O(e \times p \times size(C))$, where C is the ready list of the tasks. Therefore, for large graphs the complexity remains $O(e \times p)$.

V. Experimental Results

This section shows the application of TSRS and METS to HEFT algorithm. The comparison metric used for evaluating the schedule's length is speedup (Eq. 5). The speedup value for a given graph is computed by dividing the sequential execution time (i.e., cumulative computation costs of the tasks in the graph) by the parallel execution time. The sequential execution time is computed by assigning all tasks to the HCCP; if the HCCP is a multi-core processor, then the numerator of Eq. 5 refers to max-thread implementations.

$$Speedup = \frac{min_{p_j \in P}\{\sum_{t_i \in V} w_{(i,j,k)}\}}{makespan} \quad (5)$$

The simulation gain is given by (Simulation gain = number of simulations in total / number of simulations performed), where the numerator is given by $((\sum_{i=1}^{P} c_i + co) \times tasks)$, where P is the number of multi-core processor groups, c_i is the number of group i cores and co is the number of coprocessor groups.

A. Hardware (HW) Infrastructure

The HW infrastructure used consists of 9 different groups of processors (6 multi-core processor and 3 coprocessor groups), 3 common processors in each group (27 processors in total) and 6 cores per CPU at maximum. The groups of processors are sorted in increasing computational capability (CC), i.e., $(w_{t,9,1} \leq w_{t,8,1} \leq ... \leq w_{t,1,1})$. The HW infrastructure is described by $D.P(9)$, $C.P(3)$, $cores(6)$ arrays, giving the number of different processors, common processors and cores, respectively. So, for instance, the HW infrastructure described by $\{D.P(0,0,0,1,1,1,1,0,0)$, $C.P(0,0,0,1,2,3,1,0,0)$ and $cores(0,0,0,2,4,6)\}$, refers to one 2-core CPU of type4, two 4-core CPUs of type5, three 6-core CPUs of type6 and one coprocessor of type7. The coprocessors are of higher CC than CPUs and therefore they always refer to processors with number 7, 8 and 9. Moreover, to make the HW infrastructure more realistic, we assume that $(w_{t,7,1} \leq 5 \times w_{t,6,1})$.

B. Random graphs and computation/communication costs

We have evaluated our work to 14580 random generated application graphs. For this purpose, we used the synthetic DAG generation program Daggen [17] with five different parameters defining the DAG shape:

- n: number of DAG nodes, $n = [50, 100, 200, 300]$
- fat: this parameter affects the height and the width of the DAG, $fat = [0.2, 0.5, 0.8]$
- density: determines the number of edges between two levels of the DAG, $density = [0.2, 0.5, 0.8]$
- regularity: determines the uniformity of tasks in each level, $regularity = [0.2, 0.5, 0.8]$
- jump: indicates that an edge can go from level l to level $l + jump$, $jump = [1, 2, 4]$

To obtain the random computation and communication costs, the following parameters have been used:

- CCR: Communication-to-Computation Ratio: ratio of the sum of the edge weights to the sum of the node weights on p_{ref}, $CCR = [0.1, 0.2, 0.5, 1, 2, 5, 10]$
- β_w (Range percentage of computation costs among different tasks for p_{ref}): A high value implies wider computation costs among tasks while a low value implies narrower costs. β_w is given by the following formula where $\overline{w}$ is the average computation cost of the DAG and is selected randomly, $\beta_w = [0.5, 1, 1.5]$

$$\overline{w} \times (1 - \frac{\beta_w}{2}) \leq w_{t,p_{ref},1} \leq \overline{w} \times (1 + \frac{\beta_w}{2}) \quad (6)$$

- β_c (Range percentage of communication costs among the edges of the DAG): A high value implies wider communication costs among different edges while a low value implies narrower costs. β_c is given by the following formula where $\overline{c}$ is the average communication c value of the DAG and $\overline{c} = \overline{w} * CCR$. $\beta_c = [0.5, 1, 1.5]$

$$\overline{c} \times (1 - \frac{\beta_c}{2}) \leq c_{i,j} \leq \overline{c} \times (1 + \frac{\beta_c}{2}) \quad (7)$$

The computation costs for the other processors are generated according to the computation costs on p_{ref}. The computation costs of the remaining processors (p_i) are random values within the following range: $w_{t,p_{ref},1} \times R(i,1) \leq w_{t,i,1} \leq w_{t,p_{ref},1} \times R(i,2)$, where R=[2,2.5; 1.8,2; 1.4,1.5; 1.2,1.3; 1.05,1.15; 1,1; 0.12,0.2; 0.08,0.18; 0.05,0.15]. Regarding multi-thread computation costs, we have used random realistic speedup range values, i.e., $w_{t,i,f} = w_{t,i,1} \times speedup(f)$, where the speedup value is a random value within the following range $(1.1, 1.9)$,$(1.2, 2.8)$,$(1.3, 3.7)$,$(1.4, 4.5)$,$(1.5, 5.4)$, for $(2, 3, 4, 5, 6)$ threads, respectively. The speedup function is non-decreasing, i.e., $(w_{t,i,f1} \leq w_{t,i,f2})$, where $f1 \succ f2$.

1) Evaluating TSRS: In this Subsection, TSRS is evaluated. The results are illustrated by using boxplots in Matlab; on each box, the central red line indicates the median value and the displayed value shows the mean.

The ('Sim','ins.') in the x-axis of Fig. 1 indicate simulation gain and insertion policy, respectively. In Fig. 1, 972 different DAGs have been used (all different fat, regularity, density and jump combinations) with $n = 100$,$CCR = [0.1, 0.5, 2, 10]$, $\beta_w = \beta_c = [0.5, 1, 1.5]$ as well as several processor configurations. The '4P' indicates 4 different single-core processors. The TSRS makespan is approximately the same as that of the standalone HEFT, in all cases. Furthermore, both HEFT and TSRS perform better by using

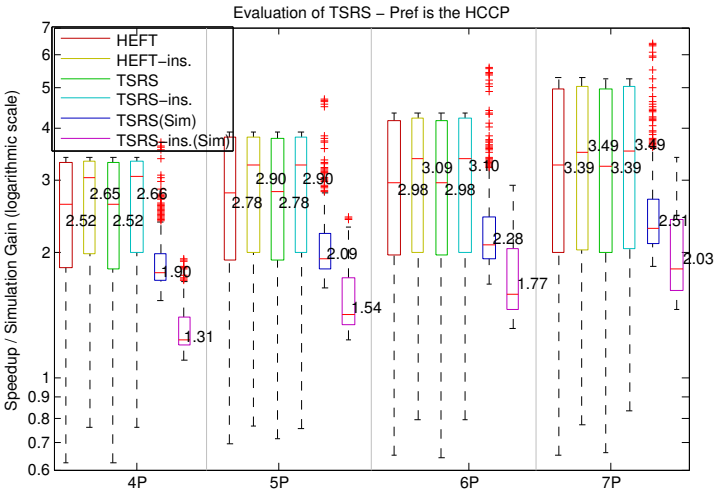

Figure 1. Evaluation of TSRS (972 different DAGs)

the insertion scheduling policy but the gains are small. By using the insertion scheduling policy lower simulation gain values occur because in that case the number of computation costs needed is higher.

2) Evaluating METS with TSRS: In this Subsection, METS with TSRS is evaluated (Fig. 2). We have evaluated METS without using the insertion scheduling policy because the makespan improvement is not significant comparing to the simulation loss. In this Subsection, $p_{ref} = 6$ in all cases; in METS the p_{ref} is always the CPU with the maximum number of cores (and ideally with the highest CC too). HEFT algorithm assumes rigid (non-moldable) tasks and therefore for a comparison to be made, we have implemented HEFT to use ST CPU implementations (SHEFT) and max-thread CPU implementations (MHEFT). Unlike our method, SHEFT and MHEFT use the insertion scheduling policy.

In Fig. 2, METS with TSRS is evaluated for all the parameter combinations in Subsection 5.2 (14580 DAGs) and six different HW configurations. The top figure in Fig. 2 refers to HW platforms where only multi-core CPUs are used, while the bottom refers to both CPUs and coprocessors. When only CPUs are used, the heuristics given in Subsection IV-B perform very well and give significant speedup values. In the first HW configuration, where the number of the available processors is small, SHEFT performs much better than MHEFT, while on the last is exactly the opposite; by increasing the number of the processors, MHEFT outperforms SHEFT as is uses processors with higher CC. Our method provides better makespan values at all cases. As it was expected, by providing more processors, all the three methods give better makespan values. Moreover, the higher the number of the processors, the higher the simulation gain as the lower CC processors are more likely to be excluded from the SL.

Regarding the bottom figure in Fig. 2, when very fast

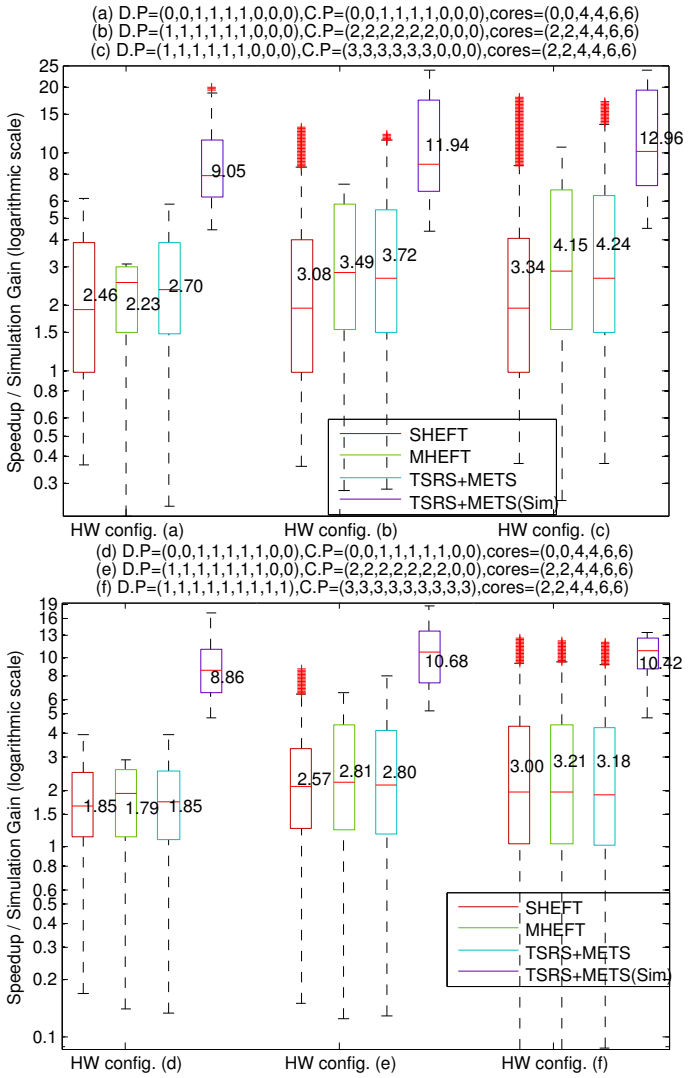

Figure 2. Evaluation of METS with TSRS (14580 different DAGs)

coprocessors are used too, METS with TSRS gives similar schedule lengths to the best of SHEFT and MHEFT, i.e., similar makespan values to SHEFT/MHEFT when the number of processors is small/high, respectively. The higher the number of the coprocessors, the less the makespan gain of our method. The reason lies in the fact that HEFT is a greedy algorithm as it always chooses the processor giving the minimum EFT value; therefore, if the coprocessors are many times faster than the CPUs, they never become idle and push aside the CPUs; thus, most of the tasks are scheduled on the coprocessors. This is why all three methods give close makespan values in cases (d) and (f), where the number of coprocessors is high comparing to the CPUs. In the last case (f), where there are nine coprocessors, most of the tasks are scheduled on the coprocessors. Still, the proposed method follows the trend of the best of the two. As far as the simulation gain is concerned, it is higher when no coprocessor exists. In this case, p_{ref} and not a coprocessor is the fastest and the most preferable processor and thus most of the tasks are scheduled on p_{ref} whose computation costs have already been computed in the initialization phase. On the other hand, when coprocessors exist, most of the tasks are scheduled and thus simulated on the coprocessors ($p_{ref} \succ 6$) while $p_{ref} = 6$ and as a consequence a larger number of extra simulations is required.

C. Real World Applications

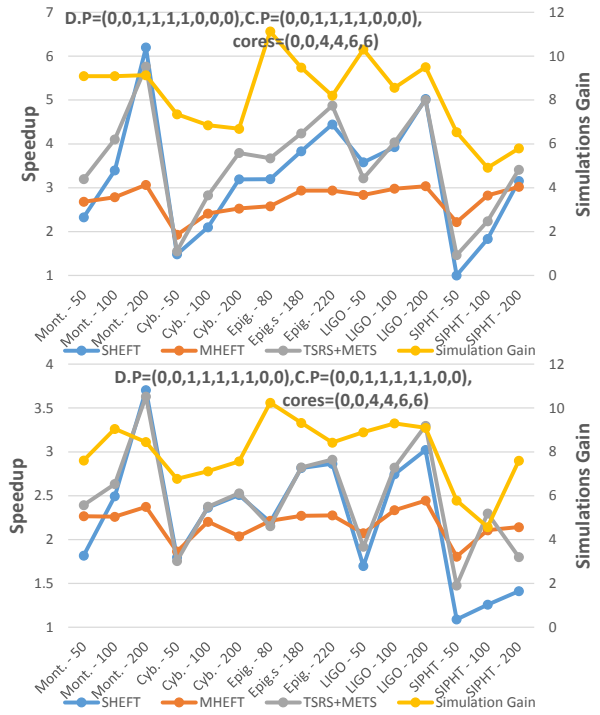

Figure 3. Evaluation of METS with TSRS, for 5 real world applications

TSRS and METS have been evaluated to Montage, CyberShake, Epigenomics, LIGO and SIPHT real world applications [18] [19]. We have used small, medium and large graphs for each one of the 5 applications (from 50 up to 200 tasks, Fig. 3) as well as real communication and computation costs for $w_{t,p_{ref},1}$, taken from [18] [19]. The computation costs for the other processors have been selected as random values within a range as in Subsection V-B. As in Fig. 3, when no coprocessor is used, METS performs better for the reason explained in the previous subsection. Moreover, SHEFT performs better than MHEFT when the number of processors is low and vise versa. Last, Epigenomics and SIPHT are less scalable.

VI. Conclusions and Future Work

We have presented two new methods for effective task scheduling in HCS. Although we have showcased both methods to HEFT, both methods are applicable to several different algorithms.

TSRS modifies HEFT's processor selection phase in order to discard all the processors which cannot minimize the heuristic cost function, regardless of their computation costs; this way, the DAG computation costs required by HEFT become limited. The insertion scheduling policy is not preferred as the makespan improvement is not significant comparing to the simulation loss.

METS refers to low complexity heuristics finding which tasks are going to be split into multiple threads as well as the number of threads used, without requiring all the computation costs in the DAG. In this paper, METS is applied together with TSRS, in order to optimize for both scheduling time and length. We evaluated METS without using the insertion scheduling policy, as the makespan improvement is not significant comparing to the simulation loss. Standalone METS gives better makespan values.

In our future work, we aim to develop a tool that takes OmpSs (Barcelona Supercomputing Center programming model) C/C++ code as input and by using TSRS and METS, it outputs a good quality schedule in low time. We use OmpSs as it extends OpenMP with new directives to support GPUs and FPGAs.

Acknowledgment

This work is partly supported by the European Commission under H2020-ICT-20152 contract 687584 - Transparent heterogeneous hardware Architecture deployment for eNergy Gain in Operation (TANGO) project.

References

[1] E. Ilavarasan and P. Thambidurai, "Low complexity performance effective task scheduling algorithm for heterogeneous computing environments," *Journal of Computer Sciences*, vol. 3, pp. 94–103, 2007.

[2] H. Arabnejad and J. G. Barbosa, "List scheduling algorithm for heterogeneous systems by an optimistic cost table," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 3, pp. 682–694, Mar. 2014.

[3] H. Topcuouglu, S. Hariri, and M.-y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, pp. 260–274, Mar. 2002.

[4] T. Hagras and J. Janecek, "A simple scheduling heuristic for heterogeneous computing environments," in *Proceedings of the Second International Conference on Parallel and Distributed Computing*, ser. ISPDC'03, 2003, pp. 104–110.

[5] E. Ilavarasan, P. Thambidurai, and R. Mahilmannan, "High performance task scheduling algorithm for heterogeneous computing system," in *6th International Conference on Algorithms and Architectures for Parallel Processing*, ser. ICA3PP'05, 2005, pp. 193–203.

[6] R. Bleuse, S. Hunold, S. Kedad-Sidhoum, F. Monna, G. Mounié, and D. Trystram, "Scheduling Independent Moldable Tasks on Multi-Cores with GPUs," *IEEE Transactions on Parallel and Distributed Systems*, p. 14, 2017.

[7] M. I. Daoud and N. Kharma, "A high performance algorithm for static task scheduling in heterogeneous distributed computing systems," *Journal of Parallel and Distributed Computing*, vol. 68, no. 4, pp. 399 – 409, 2008.

[8] S. Hunold, "One step towards bridging the gap between theory and practice in moldable task scheduling with precedence constraints," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 4, pp. 1010–1026, 2015.

[9] M. Amaris, G. Lucarelli, C. Mommessin, and D. Trystram, "Generic algorithms for scheduling applications on hybrid multi-core machines," in *23rd International European Conference on Parallel and Distributed Computing (EuroPar)*, Santiago de Compostela, Spain, 2017.

[10] T. N'Takpé, F. Suter, and H. Casanova, "A Comparison of Scheduling Approaches for Mixed-Parallel Applications on Heterogeneous Platforms," in *6th International Symposium on Parallel and Distributed Computing - ISPDC 2007*, Hagenberg, Austria, Jul 2007, pp. 35–42.

[11] S. Kedad-Sidhoum, F. Monna, and D. Trystram, "Scheduling Tasks with Precedence Constraints on Hybrid Multi-core Machines," in *IPDPSW 2015 - IEEE International Parallel and Distributed Processing Symposium Workshop*, Hyderabad, India, 2015, pp. 27–33.

[12] C. Boeres, J. V. Filho, and V. E. F. Rebello, "A cluster-based strategy for scheduling task on heterogeneous processors," in *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing*, ser. SBAC-PAD '04, 2004, pp. 214–221.

[13] L. F. Bittencourt, R. Sakellariou, and E. R. M. Madeira, "Dag scheduling using a lookahead variant of the heterogeneous earliest finish time algorithm," in *Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, ser. PDP '10, 2010, pp. 27–34.

[14] H. Zhao and R. Sakellariou, "An experimental investigation into the rank function of the heterogeneous earliest finish time scheduling algorithm," in *9th Euro-Par Conference Klagenfurt, Austria, August 26-29*, 2003, pp. 189–194.

[15] S. Baskiyar and P. SaiRanga, "Scheduling directed a-cyclic task graphs on heterogeneous network of workstations to minimize schedule length," in *Proc. Int'l Conf. Parallel Processing Workshops*, vol. 2003, 11 2003, pp. 97– 103.

[16] C. Hui, "A high efficient task scheduling algorithm based on heterogeneous multi-core processor," in *2nd International Workshop on Database Technology and Applications (DBTA)*. IEEE, 2010.

[17] F. Suter, "Daggen: A synthethic task graph generator," https://github.com/frs69wq/daggen.

[18] G. Mehta and G. Juve, "Workflow generator," https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator.

[19] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi, "Characterizing and profiling scientific workflows," *Future Gener. Comput. Syst.*, vol. 29, no. 3, pp. 682–692, Mar. 2013.